Denys Natykan

Efficient asymmetric cryptography for RFID access control

Computer Science Tripos – Part II

St John's College

May 19, 2017

Proforma

Name:	**Denys Natykan**
College:	**St John's College**
Project Title:	**Efficient asymmetric cryptography for RFID access control**
Examination:	**Computer Science Tripos – Part II, May 2017**
Word Count:	**TODO**
Project Originator:	Dr Markus Kuhn
Supervisor:	Dr Markus Kuhn

Original Aims of the Project

The goal of my project is to have a working prototype of an RFID smart-card system that uses asymmetric-key cryptography. The cryptographic protocol should be either designed by me or based on an existing protocol, protect users against impersonation, be secure against replay attacks and be able to sustain attempts to brute force the keys. Authentication of the card must not take longer than one second. In the end I should have a Command Line Interface (CLI) application that allows initialisation and reprogramming of the cards. The CLI must also implement an emulation of the card authentication process with a door controller.

Work Completed

I have created a working implementation of a door access control system based on a asymmetric cryptography protocol. The designed protocol is protected from replay, brute-force, impersonation and cloning attacks. It authenticates the card in around 170 ms, meaning my project was a success.

Special Difficulties

None

Declaration

I, Denys Natykan of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed Denys Natykan

Date 19th May 2017

Contents

List of Figures

Chapter 1

Introduction

A smart-card is an electronic data storage system, possibly with additional computing capacity, which for convenience is incorporated into a plastic card the size of a credit card.[1] The first ever mass-produced smart-cards where one-time phone card that stored the balance of the user. The balance could only be reduced after the card initialization at manufacturing stage, so when the user ran out of balance, he had to throw away the old card and buy a new one. Since then smart-card technology has become ubiquitous, and part of many aspects of our everyday life. Imagine travelling on the tube, paying for your groceries, or buying clothing in a shop. RFID technology can be encountered in each of these activities.

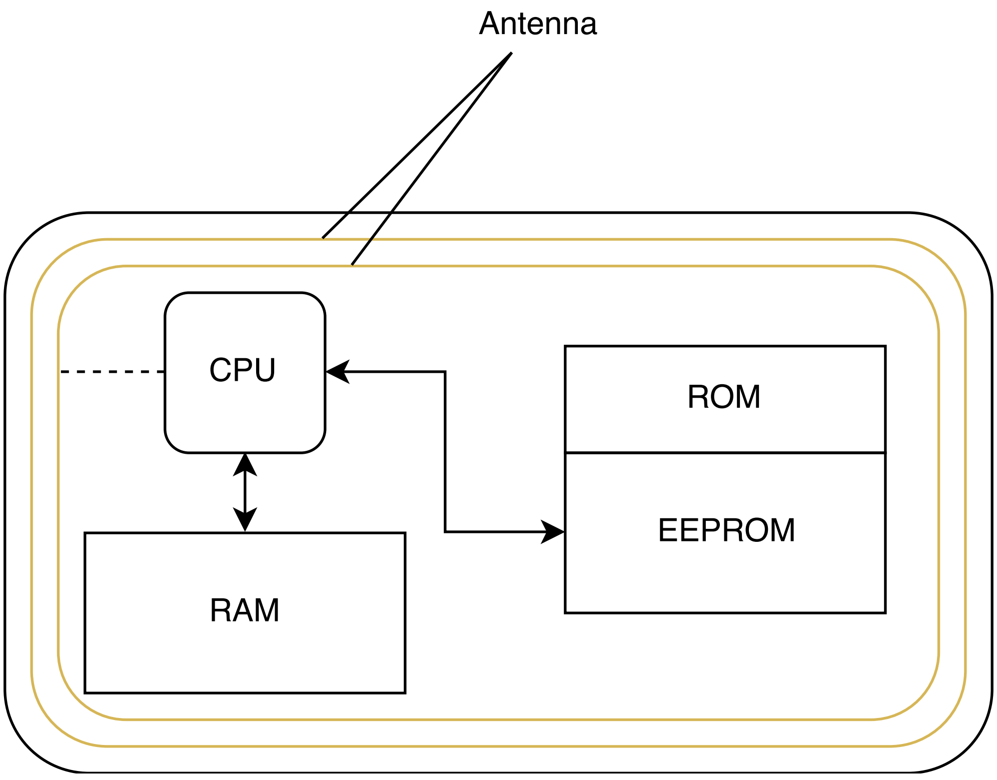

Figure 1.1: Scheme of microprocessor smart-card

Inside of the contactless smart-card there are several loops of wire that both serve as an antenna for wireless communication and to wirelessly power the card through inductive coupling (when the card is exposed to the Card Accepting Device (CAD) the current is induced in the coil, which powers up the whole card including the transmitter, that card uses to communicate with a CAD). One of the main reasons for growth of popularity of smart-cards is the claimed security that they provide. Most of the time, smart-cards are actually more secure than any of the potential technological substitutes, due to the fact that data stored on the smart-card can be physically protected from undesired access and manipulation by dedicated hardware. However it is a continuous arms race between the attacker and smart-card industry, which means that even the most trusted systems can become victims and be compromised one day.

The MIFARE Classic smart-card system, produced by NXP Semiconductors, is arguably one of the least secure cryptographic card systems still in use. Vulnerabilities of MIFARE Classic are particularly important to the University of Cambridge, which still uses MIFARE Classic technology in every University card.

1.1 Asymmetric vs symmetric cryptography

To consider possible alternatives to the MIFARE Classic system, one needs to understand the cryptographic algorithms employed to protect the secrets stored on smart-cards. MIFARE Classic and many other smart-card systems are based on symmetric cryptography, also called secret-key cryptography. These schemes require both the sender and the receiver to share the same secret key, in order to be able to communicate. As with any solution, this one has its advantages and disadvantages.

On the positive side: secret-key cryptography is faster than asymmetric cryptography, and uses less computer resources. However, it has one major drawback that has been proven to be a game changer: all elements of the system need to share same secrets in order to communicate with each other. Hence if one device is compromised, the whole system becomes vulnerable. For example, in a system, such as a university door access control, readers and cards are likely to contain multiple shared secrets (e.g. a card will contain a shared secret key for each department it has access to). So if one card gets compromised all the departments it has access to are now accessible to an attacker. The fundamental solution to this problem is to use an asymmetric cryptography as the basis of a smart-card security system.

Asymmetric cryptography is based on two keys, rather than one. The first key the public key, is transmitted openly to everyone who wants to communicate with the host. The hosts private key is kept secret and never leaves the hosts device. This ensures that if someone in some way gets hold of the hosts secret key, the only device that will be compromised is host. So to compromise the whole system, every device must be hacked. Considering the fact that the secret key is never actually transmitted to anyone and is not used for creating any ciphertext, it is very difficult to imagine a scenario during which someone gets hold of it. Asymmetric cryptography is more computationally intensive than symmetric, which in the context of low computing power of smart-cards plays a significant role, and might create some complications.

1.2 Memory cards vs microprocessor card

All smart-cards can be divided in two types: memory cards and microprocessor cards. Memory cards, can only store data, and are not able to do any proper processing of data. All their functions are limited by a small circuit with a few pre-programmed instructions. Microprocessor cards on the other hand contain a small CPU, which means that they have multifunctional capability and that applications can be added to and deleted from the card after it has been manufactured. As I am dealing with asymmetric cryptography in my project I will only be dealing with microprocessor smart-cards. Almost all current models of microprocessor smart-cards have built-in support for some cryptographic algorithms.

1.3 University smart-card system - MIFARE Classic

As the initial stage of my preparation I looked at the system that is currently used in the University of Cambridge - MIFARE Classic, concentrating on the problems that are associated with it. This was needed to better understand the requirements and success criteria for my prototype.

1.3.1 Historic background

MIFARE Classic technology was mainly sold by NXP Semiconductors (a spin-off of Philips Electronics) starting from 2006. When it first appeared on the smart-card market it was claimed to be secure enough to carry out contactless payments. The cryptographic algorithm that was used in the MIFARE Classic cards is called CRYPTO-1, and was designed by the card manufacturer specifically for this model of card. Producers decided to keep the algorithm of CRYPTO-1 secret, which is called security by obscurity. In December 2007 two German researchers[2] managed to partially reverse engineer the algorithm of CRYPTO-1, and from that they were able to demonstrate some weaknesses of the algorithm. In March 2008 a research group from Radbond University managed to completely reverse engineer the architecture of the card and therefore the CRYPTO-1 algorithm. This group then won a case against NXP Semiconductors for the right to publish their work. Since then, a number of attacks have been designed on the system, the most effective of which will be discussed later.

1.3.2 Structure and memory organisation of MIFARE Classic cards

MIFARE Classic cards are memory smart-cards. The memory of all cards is split into sectors that consist from blocks. Each sector has two secret keys: key A and key B. These keys play the role of a shared secret, to prove that a device is authorised to read data in the sector it proves that it knows one of two keys. The keys are stored in the final block of each sector, which is called "sector trailer", along with access bits for this sector. The reason there are two keys rather than one is to give different access permissions to different parties. For example knowing key A gives permission to read the data in the sector, and knowing key B gives permission to read the sector and write data to the sector. The data that belongs to a sector is usually logically connected. For example one sector can represent the door access of the owner of the card within a particular college. The first block of sector 0 always contains Unique Identifier (UID) of the card, that is written to the card during its production and cannot be changed. In university cards the first sector also contains the CRSID and student number of the owner. There exist two variations of the card: 1k and 4k, which correspond to the size of memory available on the card. All of the Cambridge University cards are 4k cards. To access a block on the card a CAD needs to complete challenge-response authentication process, using one of the keys for the sector, that contains the desired block.

1.4 Introduction to Java Card

The Java card platform provides an advanced framework for developing applications for microprocessor smart-cards. Its multiple layer security structure and built in support for atomic transactions together with standard Java language features provide a secure and isolated environment for each application that runs on the card. Java Card platform also allows for customization after the cards issuance, as multiple applets can be installed from different vendors dynamically at any point of the cards lifetime. Being a subset of Java language Java Card lessens security risks is several ways: prohibition of pointer arithmetic excludes possibility of attackers being able to access the memory in an unauthorised manner, strict definition of Java ensures that there are no ambiguities about the sequence of instruction execution or general virtual machine behaviour. These features made the Java Card platform a clear choice for my project.

Chapter 2

Preparation

2.1 Most notable weaknesses of and attacks on MIFARE Classic

2.1.1 Key length and brute-force attack

Keys of a MIFARE Classic card are only 48 bits of length, which makes a brute-force attack (exhaustive search of the key) on the card feasible. The calculations show that it would take around 2 years to guess the key to one sector on single core CPU. However if the malicious party uses a FPGA, the time of the search of each key can be shortened to a couple of hours. Hence we can see that a brute-force is possible in the case of MIFARE Classic, however a number of faster attacks exist.

2.1.2 Prediction of nonce generated by a card

Every message exchange between a card and a CAD starts with mutual authentication. Mutual authentication lets each party know that the other side is who it is supposed to be, without ever exchanging any secrets. As the part of the handshake, the smart-card generates a pseudo-random byte (nonce) and sends it to the CAD. The whole point of a nonce is that it should be unpredictable and indistinguishable from a truly random byte sequence. However MIFARE Classic pseudo-random number generator (PRNG) starts from same state after each reset (power up). It also has a known period of 618 ms, so by measuring the time that has elapsed since the power up of the card, the attacker can predict the nonce generated by the card. This vulnerability resulted in appearance of "Nested attack" on MIFARE Classic cards.

2.1.3 Nested attack

A lot of MIFARE Classic card systems use default and well known keys for some sectors (e.g. University cards use not one but a few default keys for its sectors). A nested attack can only be performed if at least one of the sector keys is a default key or is already known to the attacker. As the first step of the attack, the malicious party authenticates to the block with default/known key and reads card's nonce. After the analysis of the retrieved nonce, the attacker is able to restore the state of the PRNG at the moment of the nonce generation. Due to the fact that the period of the PRNG is known, the attacker is now able to predict card generated nonce, by carefully timing his or her requests, and predicting the state of the PRNG at the moment of nonce generation. The ability to predict a card's nonces helps attacker to obtain key stream for every sector on the card.

2.1.4 Parity bit weakness and Dark Side attack

The parity bit is used for error detection during data transmission. It is a bit that is added to a binary string to ensure that the total number of 1s in the string is even (or odd, depending on agreement). The ISO 14443-A standard requires that in every byte exchanged between a card and a reader one bit is always reserved as a parity bit. MIFARE Classic implements ISO 14443-A. The secure way to add parity bit, is to add it after the transmission data has been encrypted, so the parity bit does not leak any information about the plaintext. However, a MIFARE Classic implementation of the standard adds the parity bit to the transmission data before encryption. And what is most important, is that it allows an attacker to distinguish between the case when both the key to the sector is wrong and the expected parity bits are wrong, and the case when the key is wrong, but the expected parity bits are right. This fact can be exploited to retrieve some bits of the sector key. Combined with the ability to predict card's nonce, an attacker is able to slowly, bit by bit, reconstruct the key to the sector, by making thousands of requests that fail and analysing the card's response. This attack is called "Dark Side"[3] attack.

2.1.5 Cloning attack

Each MIFARE Classic card has Unique Identifier (UID) stored in block 0 of sector 0 on the card. The UID is burnt on the card in the factory and cannot be modified after that. This protects cards from cloning, as even if all of the contents of the card is cloned to the fresh card, the UID of two cards will not be the same. Hence the new card will not work. However there exist cards that fully replicate the internal structure of MIFARE Classic, with the only difference that they have a changeable UID. They are called UID changeable card and they allow anyone to clone any MIFARE Classic card provided they only have a PC/SC tag reader and a UID changeable card. This means that an attacker can impersonate any identity on the system, provided he or she have been close enough to the victims card to make a clone.

2.1.6 Attack combination used

Nested attack and Dark side attack can be combined together to create an ultimate MIFARE Classic hacking tool, that hacks the card in the fastest known way. Initially the card tested for usage of the default keys. If at least one default key is found the nested attack is performed. If the card does not use any default keys the Dark Side attack is performed to retrieve key for any sector, and then nested attack is performed, using the sector with the known key as a starting point.

2.2 Communication between card and a reader

The unit of communication between the Card Accessing Device (CAD) and smart-card is known as the Application Protocol Data Units (APDUs), the structure and contents of which is defined by ISO/IEC 7816-4. The master-slave model is implemented in any smart-card system. This means that card awaits instructions from the CAD that come in form of Command APDUs. It then processes the command and replies with results using Response APDU. Figure 2.1

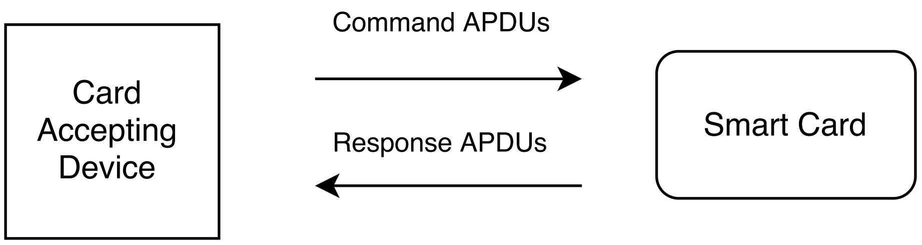

Figure 2.1: APDU exchange between CAD device and smart-card

The APDU message structure is specified by ISO 7816-4, which restricts any APDU to the two possible structures: one that applies to APDUs from CAD to a card (Command APDU), and second one that applies to APDUs from card to a CAD (Response APDU).

2.2.1 Command APDU

Command APDU must consist of mandatory header, and optional body. Header consists of 4 fields each 1 byte long: CLA class of instruction (identifies the category of APDU), INS - instruction code, P1 and P2 two parameters (provide further information on the instruction). After the mandatory APDU header goes optional body of the command. It consists of 3 fields: Lc, Data Field, and Le. Lc field is 1 byte long and is used to specify the size of data that must follow in Data Field. Data Field can vary in size with one condition that it must correspond to Lc field. Last field Le, is used if the host wants to specify the length of the expected response APDU. Le field is not mandatory.

Mandatory header				Optional body		
CLA	INS	P1	P2	Lc	Data Field	Le

Figure 2.2: Structure of command APDU

2.2.2 Response APDU

Response APDU consists of two parts as well: an optional body and a mandatory trailer. Body comprises Data Field which can be of variable size and is used to send data response form card to CAD. Trailer consists of two mandatory fields, each 1 byte long, called SW1 and SW2. SW1 and SW2 combined give Status Word, which indicates the result of executing Command APDU (e.g. 0x9000 indicates complete and successful execution of the command and 0x6A89 indicates File already exists error).

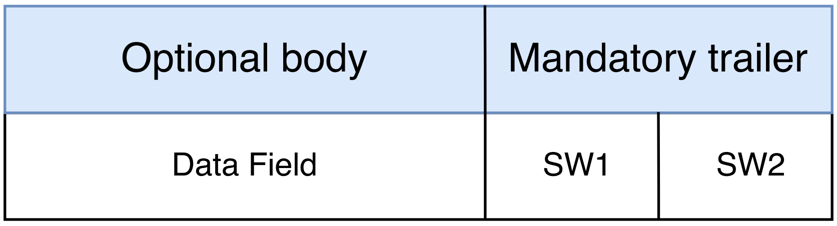

Figure 2.3: Structure of response APDU

2.3 Java Card technology overview

2.3.1 Java Card language subset

Because of relatively small size of the memory available on the card, the Java Card platform supports only a subset of the Java language. This subset is customized and very carefully chosen, so that it can support all the functions that are needed to write a smart-card application and yet remain lightweight. We will not discuss all the differences and limitations of Java Card language compared to Java, as this can easily take the word count of the whole dissertation. However we will take a look at the most crucial aspects, as they will appear in example code or as they are mentioned.

2.3.2 Java Card Virtual Machine

The main difference between Java Virtual Machine (JVM) and Java Card Virtual Machine (JCVM) is in the fact that JCVM is implemented in two separate pieces: off-card converter, that runs on PC or workstation and on-card byte code interpreter (2.4). Together these 2 pieces implement all the virtual machine functions: converter loads all the Java class files in the package, preprocesses them and outputs Converter Applet file (CAP file). It also produces an export file, that represents a public API of the converted package. The CAP file is then loaded on to the Java smart-card, where interpreter executes it.

2.3.3 Java Card applet

A running application in JCRE environment is called an applet. Every running applet in an instance of the applet class. Every applet class extends `javacard.framework.Applet` Running applet is a persistent object, and as any persistent object, once created, it lives throughout the lifetime of the card. As said earlier Java Card platform allows for a multi-application environment. Hence each applet needs to be uniquely identified, so the host can refer to it. This is done through the use of Application Identifiers (AIDs). To prevent occurrence of race conditions or any other problems that can arise in other

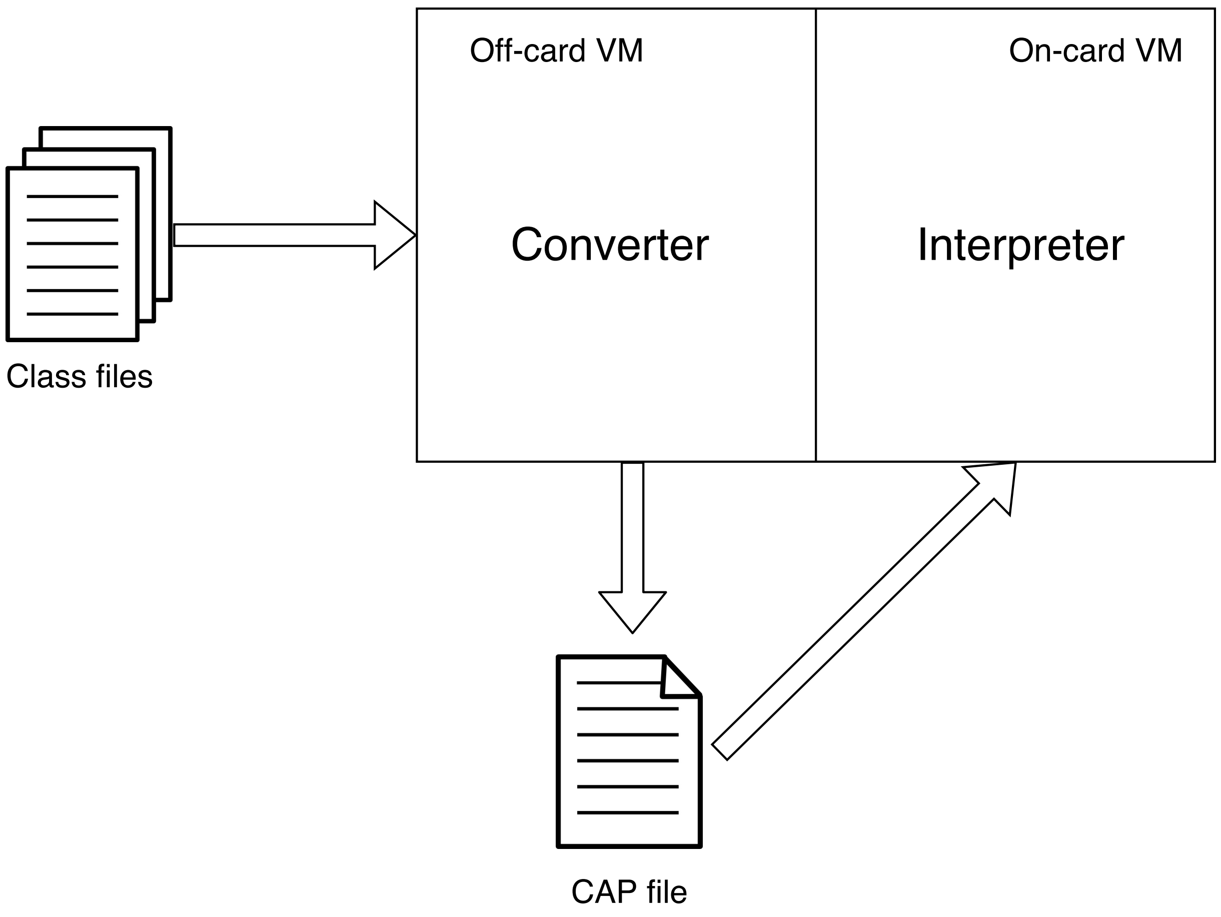

Figure 2.4: Java Card Virtual Machine

non-isolated system and to make the Java Card language more lightweight the Java Card platform only supports single thread computation. Hence only one applet can run at any particular moment.

2.3.4 Applet installation and execution

After the on-card interpreter executes CAP files of the applet, and packages that define the applet are properly linked with other packages on the card, an instance of the applet is created and registered within the JCRE. As the JCRE is a single thread environment, only one applet can be run at a time. After the applet is installed it can be in one of two states: active or inactive. Applet state machine is illustrated in Figure 2.5. Straight after the installation the applet enters the inactive state, and can only enter active state if explicitly selected by the host.

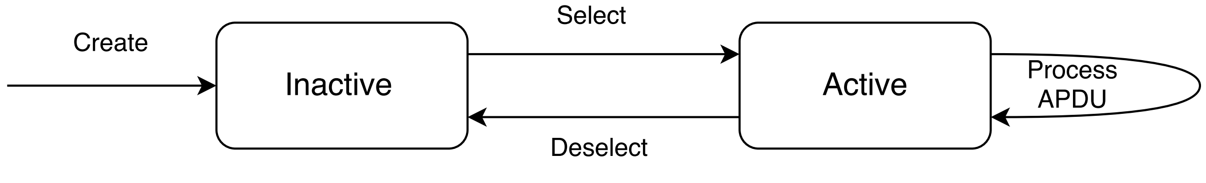

Figure 2.5: Java Card applet state transition diagram

Java Card applets are reactive, which means that they wait for the command from the host, execute it and send the response. This command-response exchange continues until the applet is deselected or the card is cut from the power. After both deselection and power cut, the applet remains inactive until it is selected again.

2.3.5 Memory organisation on Java Card smart-card

There are 2 types of memory that can be used to store objects created by a Java Card applet: transient Random Access Memory (RAM) and persistent Electrically Erasable Programmable Read-Only Memory (EEPROM). Apart from behaving differently after the power down of the card, the two memory types also differ in size available and in speed of writes. Usually the size of RAM available on the card is a few times smaller than the size of EEPROM (J3A040 has 40 kilobytes of EEPROM and only ~6 kilobytes of RAM), hence the RAM must be allocated more carefully, so it does not get filled during the card operation. RAM is about 1000 times faster to write to than EEPROM.

This differences result in different use for each type of the memory. For example a public signing key of the terminal that needs to be stored in the object on the card, and remains unchanged for a long period of time, is recommended to be kept in a persistent object in EEPROM. However, if we are talking about temporary buffer that is used as a helper memory during calculation of `HMAC-SHA-256`, and does not need to be persistent form one power-up to another, it is recommended to use RAM.

Persistent objects

Persistent objects are created by using `new` operator. Once the persistent object is created, it exists and takes allocated space in the memory until it is garbage collected (not all java cards implement garbage collection). Any update to any field of the persistent object is atomic. This means that if during the update the power is lost, or any other error occurs the field is restored to its previous value, hence it is protected from data inconsistencies. EEPROM can only do certain number of rewrites, so every the object is created or varied in persistent storage, the card's lifetime.

Transient objects

A transient object can only be created by using the Java Card APIs. Any update to the field of the transient object is non-atomic, that is in case if the power is gone during an update the value of the field is not restored to the previous one (when the power is gone, all the data is wiped from the transient object anyway). The above mentioned properties of the transient objects make them perfect for using as a storage for small amounts of temporary applet data, that is updated frequently and does not need to be preserved across CAD sessions.

It is worth mentioning, that transient objects are not themselves temporary once created, transient object is preserved from one CAD session to another, however, its contents are not. Every time the card is removed away from the power source the contents of all the transient objects get wiped clean. As with the persistent objects, once the transient object is created it will be allocated space in the memory up until it is garbage collected. Hence an applet should create a transient object only once in its lifetime, and save its reference in non-transient field, as shown in the Figure 2.6. Every time the card is powered up, the same reference to the transient object will be used, even though its contents have been erased.

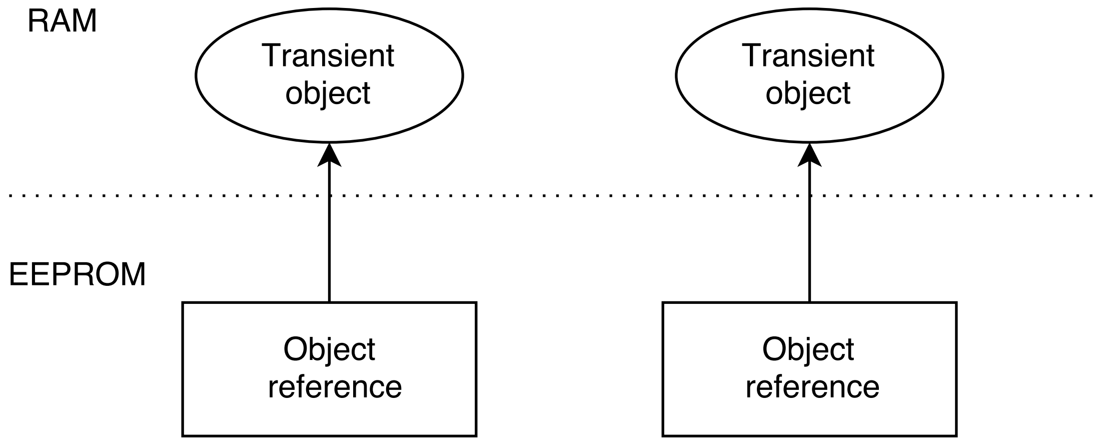

Figure 2.6: References to transient objects

2.4 Framework and tools used

2.4.1 GPShell

The Global Platform card specification is a standard for the management of the infrastructure of a smart-card[4].This management includes application installation and removal and additional management tasks that are run on the card. Global Platform can be implemented on any multi-application smart-card environment, but is mainly used with Java Card Runtime Environment, where its main functions are applet installation and deletion. GPShell is a script interpreter that communicates with the smart-card and complies to Global Platform Card Specification[5]. GPShell uses PC/SC[7] plug-in to talk to the smart-cards. GPshell is capable of establishing a secure channel with a smart-cards, list, load, delete and instantiate applets on the card. GPShell runs script files that can be written in plaintext format, and turns each command into the set of command APDUs which is sent to the card. Below is an example of the script file to install Applet.cap on the card can be seen.

```
enable_trace
establish_context
mode_211
card_connect
select -AID a000000003000000
open_sc -security 1 -keyind 0 -keyver 0 -mac_key
   404142434445464748494a4b4c4d4e4f -enc_key
   404142434445464748494a4b4c4d4e4f
```

```
install -file Applet.cap -sdAID a000000003000000 -priv 2
card_disconnect
release_context
```

- `enable_trace` lets the user see the APDUs send to the card and the responses received. It is needed for error detection and correction purposes.
- `mode_211` specifies the version of the Global Platform, the card complies to. In this case it is GlobalPlatform 2.1.1.
- `card_connect` connects to the card via a default reader.
- `select -AID a000000003000000` connects to the Card Manager, that is responsible for applet installation.
- `open_sc -security 1 -mac_key 404142434445464748494a4b4c4d4e4f -enc_key 404142434445464748494a4b4c4d4e4f` opens secure channel with with default keys. The keys for a secure channel vary depending on card manufacturer and card operating system. In this keys these are Java Card Open Platform (JCOP) standard keys, as this is the operating system that runs on JCOP J3A040.
- `install -file Applet.cap -sdAID a000000003000000 -priv 2` loads and installs an applet that is contained in `Applet.cap` in one step. The command specifies Security Domain AID (sdAID), which in this case is card manager, and privileges of the applet that is being installed.
- `card_disconnect` disconnects default reader from the card.
- `release_context` releases context that was associated with disconnected card.

2.4.2 JCIDE

Java Card Integrated Development Environment is software package designed specifically for Java Card application development. It includes Java Card emulator and debugger. The software package also included few other additional applications, such as JCAlgMaster and pyADPDUTool. JCAlgMaster that provides a functionality of cryptographic algorithms, that are commonly implemented as a part of standard library on Java Cards. It can be used for checking whether the particular algorithm was used correctly. Despite containing only a subset of all algorithms, it proved to be very useful for debugging purposes. pyAPDUTool is python wrapper for libNFC, that allows you to delete and install applications to the Java Card and send separate APDUs. Even though the graphical user interface of the tool makes it appealing at first when compared to alternatives, such as GPShell, it does not provide any logging or access to underlying APDU communications that happen during the installation/deletion process, and that is a substantial disadvantage.

2.4.3 JavaCardPro

A web-site javacard.pro[8] provides access to many open source components for building Java Card applets. The one that I found particularly valuable was the Ant task for building Java Card CAP files. It and supports all available JavaCard SDK versions. For my implementation I simply followed the detailed instructions that can be found on the website.

2.4.4 Libnfc

Libnfc is open-source C++ library for communication with various smart-cards, that supports modulations for ISO/IEC 14443. The initial plan for my framework was to write Java wrapper for Libnfc, to send all the command APDUs (e.g. application installation/deletion commands). Despite having done a lot of the initial work needed for creating the wrapper, I later discovered higher level tools like GPShell, and decided to use them.

2.4.5 Model of smart-card used

Choosing a smart-card was not an easy task, especially considering how difficult it was to find the specification for a particular card. I decided to use NXP J3A040 as it is one of the most commonly available Java Card smart-cards, that is available for fast delivery, has a reasonable price and supports all the features that were needed for my project (Java Card in compliance with ISO 14443 contactless interface standard and supporting GlobalPlatform API).

Each model of smart-card differs not only in hardware characteristics like the amount of memory available and the type of microprocessor used, but also in cryptographic algorithms implemented as a part of the standard library. Full characteristics of the card can be obtained by software, like JCAlgTest[9]. The choice of the type of `ECDSA` used in my project was mainly based on the algorithms supported by J3A040.

Chapter 3

Implementation

3.1 Java APDU wrapper

In order to abstract away from dealing with every single APDU, I created a wrapper class `CardCommunicator.java`, that provided a high-level API for talking to a card. It contains methods like `establishConnection()` that is used by the issuer terminal to initialize communication with a card, `waitAndEstablishConnection()` that is used by the door terminal in order to continuously scan for card in operating proximity of the reader and establish the connection as soon as the card is close enough.

Here is a method `selectApp(byte[] AID)` that can be used to send command to JCRE to select a particular applet on the card:

```java
 1  protected void selectApp(byte[] AID)
 2  {
 3      byte apdu[] = new byte[APDU_HEADER_LEN_LC + AID.length];
 4      System.arraycopy(APDU_SELECT, 0, apdu, 0, APDU_HEADER_LEN_NO_LC);
 5      apdu[OFFSET_LC] = (byte) AID.length;
 6      System.arraycopy(AID, 0, apdu, OFFSET_DATA, AID.length);
 7      try
 8      {
 9          ResponseAPDU r = channel.transmit(new CommandAPDU(apdu));
10          byte[] response = r.getBytes();
11          printResponse("SelectApp", true, response);
12      }
13      catch(CardException e)
14      {
15          e.printStackTrace();
16      }
17  }
```

The structure of created APDU follows `ISO7816`, which is described earlier and can be seen in Figure 2.2. `APDU_SELECT` is a byte array that contains first four bytes of the select instruction (CLA, INS, SW1 and SW2): `{(byte)0x00,(byte)0xA4,(byte)0x04,(byte)0x00}`. Lc byte is filled on `line 5`. After that the method just copies the contents of the `AID` array into the end of `apdu` array (`line 6`) and sends the command to the card (`line 9`). The response is then printed for debugging purposes.

More complex methods like `byte[] signOnCard(byte[] nonce, byte[] id)` can involve more than one APDU being sent and received from the card.

3.2 Appet development process

3.2.1 Java Card applet

A java card applet is the application written in Java programming language that meets the requirements for running inside Java Card Runtime Environment (JCRE). A running applet is an instance of an applet class that extends `javacard.framework.Applet`. Instance of an applet can be seen as the persistent object created in EEPROM memory of the card, so it lives throughout the lifetime of the card. Java Card platform supports multiple Applets and multiple instances of the same Applet to be instantiated on the card at the same card, so each applet instance needs a way to be uniquely identified. This is achieved through the use of unique Application Identifiers (AIDs).

3.2.2 Application Identifiers

The structure of the AID is specified in ISO 7816 and is used to uniquely identify the package and applets that belong to the package (Figure 3.1).

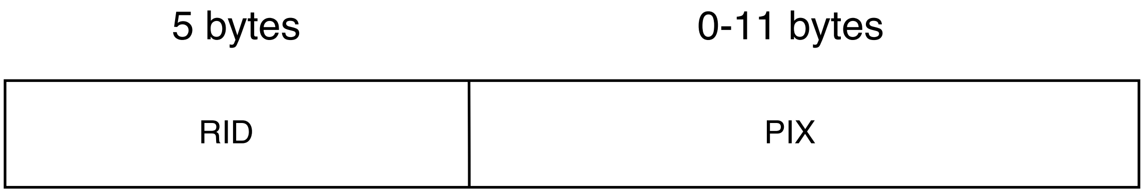

Figure 3.1: AID structure

AID consists of two parts. Resource Identifier (RID) is 5-byte long, and is assigned uniquely to each company that wants to develop Java Card applets for commercial use. ISO controls the assignment of RIDs to the companies. Second part of the AID is Proprietary Identifier Extension (PIX) and is from 0 to 11 bytes long. The companies control the assignment of PIXs to their applets themselves. An applet or package AID should be different from any existing AID on the card. However if an applet is in the particular package, than RID of the applet and the package it is in must be the same as they are coming form the same manufacturer. Even though it is not necessary, I used the convention in naming packages and applets that is similar to the naming convention used in Java, where the full name of the applet includes name of the package it belongs to. So my applet AID contains the AID of the package it belongs to (e.g. `0xD1D1D1D1D1D10101` is an AID of an applet, that belongs to the package with AID `0xD1D1D1D1D1D1`).

3.2.3 Applet installation

The process of the applet installation on the card is quite complex and not especially relevant to the applet development, as it was automated by GPShell. Hence I will only mention the bits that actually influence the code of the applet.

Install method As the last step of applet installation the on-card installer needs to create an applet instance and to register it within the JCRE. To do so the on-card installer invokes the install method.

```
public static void install(byte[] bArray, short bOffset, byte bLength)
{
        new MainApp(bArray, bOffset, bLength);
}
```

Byte array `bArray` is used to provide installation parameters to the new instance of the applet. Installation parameters start at offset `offset` into `bArray` and are of length `length`. The `install` method can be thought of as something similar to `main` method in Java language with `byte[] bArray` having the same role as `String[] args`. `install` method calls the constructor, which in turn invokes method `register()` to register the existence of the new applet instance within the JCRE. After the applet is initialized and registered within the JCRE it can be selected and run. The host can select an applet to interact with using applet's AID.

process Method When some particular applet is selected and JCRE receives an APDU from host, JCRE calls the selected applets process method. The process method is used as a dispatcher. It decodes the received APDU and calls the methods needed to provide the response to the host. An extract from my main applet's process method can be seen in Listing 3.1.

```
68 ...
69 ...
70 ...
71 public void process(APDU apdu)
72 {
73         byte buffer[] = apdu.getBuffer();
74 
75         byte ins = buffer[ISO7816.OFFSET_INS];
76         switch(ins)
77         {
78                 case INS_SELECT:
79                         break;
80 
81                 case INS_GENERATE_KEYS:
82                         break;
83 
84                 case INS_SIGN:
85 ...
86 ...
87 ...
```

Listing 3.1: Part of `process(APDU apdu)` method

The JCRE provides an applet's access to the APDU through `apdu` argument of the process method(line 71 in Listing 3.1).

3.3 Memory usage on the card

As it was mentioned earlier, when developing applets for Java Card platform, extra caution should be given to the type of memory used when storing an object. In this section I want to demonstrate a couple of examples of using different kinds of memory in my applet.

3.3.1 Examples of transient objects

Array used to return results to the main applet class

First example of a transient object is an array that is used to return the MAC key of the department to the main applet class from the persistent key storage. This buffer is likely to be rewritten very often, and writes are much faster with RAM. Also many frequent rewrites are harmful for EEPROM. We do not care whether the array loses its contents when the power is down, or not, as the data it ever stores is only relevant to the current CAD session. Hence the only correct solution in this particular case is to use transient memory. Transient object is created by invoking Java Card APIs:

```
byte[] resultAr = JCSystem.makeTransientByteArray(LEN_TEMP_BUFFER,
    JCSystem.CLEAR_ON_DESELECT);
```

Two types of transient objects can be created: CLEAR_ON_DESELECT and CLEAR_ON_RESET.

CLEAR_ON_RESET objects preserve data that they store across multiple applet selections and deselections, but not across card resets (reset signal that is sent to the card by CAD is called warm reset, removal of the card from the proximity of the CAD is called cold reset).

CLEAR_ON_DESELECT transient objects are used for storing the data that needs to be preserved only for the time during which an applet is selected. This objects will be erased as soon as the applet that they belong to is deselected or reset (both warm and cold resets automatically deselect an applet).

In our case the MAC key that is stored in the resultAr is only relevant to the authentication applet, and no other applet that is run during the same CAD session as authentication applet should need contents of resultAr. Hence we make our object CLEAR_ON_DESELECT.

Temporary buffer used for calculation of HMAC-SHA-256

HMACSHA256.java class, is used to calculate the signature on the card. This is my implementation of HMAC-SHA-256 according to RFC4868, which is looked at in some detail later in this chapter. During HMAC calculation some amount of memory is needed as a temporary buffer for data. This buffer is overwritten many times during single signature calculation, and data in this buffer is only relevant for the current CAD session. Again the only right choice of type of an object for this buffer would be transient. For the same reasons that were mentioned in discussion of resultAr I used CLEAR_ON_DESELECT:

```
byte[] tempBuffer = JCSystem.makeTransientByteArray(LEN_TEMP_BUFFER,
    JCSystem.CLEAR_ON_DESELECT);}
```

3.3.2 Examples of persistent objects

Persistent key storage for department MAC keys

Some memory is needed in order to store `MAC` key of different departments. The storage must be persistent from one CAD session to another. This is the case when we can only use EEPROM. In my card applet code storage class `KeyStorage.java` stores keys in `byte` array called storage:

```
this.storage = new byte[(short)(size*STORAGE_UNIT_LEN)];
```

ECDSA key pair

The `ECDSA` key pair that gets generated on the card needs to be persistent all the time and preserved over different CAD sessions. The keys would usually be only generated once throughout the lifetime of the card, hence this memory will mainly be read from, not written to. The type of memory that should be used in this case, is, obviously, EEPROM:

```
objECDSAKeyPair = new KeyPair(KeyPair.ALG_EC_FP,
   KeyBuilder.LENGTH_EC_FP_192);
```

3.4 APDU data field format

Sending big data (e.g. keys and signatures) from host device to the card and back in APDUs needed some kind of formatting agreement for the contents of APDU's Data Field. In some cases I needed to send up to 4 items in the same APDU (e.g CRSid, date, public `ECDSA` key and signature), and getting the sizes and offsets of each item right quickly became tedious on both sending and receiving side. Also, lack of formatting meant, that every change in size in one of the items sent in a Data Field, will cause receiver to fail to interpret the data field, unless receiver's code is changed as well.

As a solution to above mentioned problems a simple formatting scheme was designed for the cases when the data field of an APDU to be transmitted is complex. The schematic structure of the formatting scheme is displayed in Figure 3.2. As a basis for the scheme I used ideas introduced in Roughtime[10] protocol developed by Google.

I decided to make data field a map from uint32 to byte strings, so it follows a simple structure. All values are agreed to be encoded in big endian for consistence. Each such data field consists of two parts: header and payload. Both parts can have variable length depending on the size of the data field. Overall the structure of the data field designed to be as flexible as possible.

The header of data field consists of 3 parts that have a fixed order: number of tags, array of the tags, and array of the offsets. The number of tags must be the same as the size of both the tags array and the offsets array. The maximum number of tags that can be in one data field and also the total length of the data field (in bytes) is $2^{32} - 1$ (hexadecimal `0xFFFFFFFF`), due to the size of uint32. Tags are used to describe the contents of the byte string at particular offset. Tag can be any 32 bit integer, however in my project I used

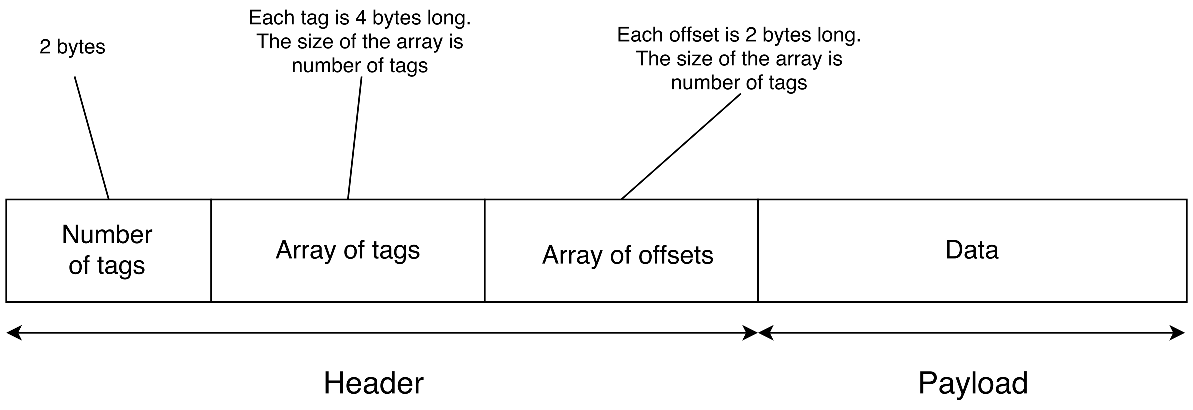
2 bytes
Each tag is 4 bytes long. The size of the array is number of tags
Each offset is 2 bytes long. The size of the array is number of tags
Number of tags
Array of tags
Array of offsets
Data
Header
Payload

Figure 3.2: Format of data field

tags, that represent 4 characters, when encoded in big endian using UTF-8 encoding. For example CRID stands for CRSid of the person, and when encoded, it turns into `0x43524944`, which is used as a tag for CRSid. Similar DATE stands for date (e.g expiry date of certificate), PKEY for elliptic curve public key and SIGN for signature.

An offset array gives an offsets for data of each tag into the data field. For example if the first element in the tags array is CRID and the first element of the offset array is 12, this means that the CRID byte string starts at the 12th byte of the data field. To figure out the length of the data, we just look into the next offset. Lets assume it is 17. Hence we know that the length of the CRID byte string is $17 - 12 = 5$. In case of the last element of the offset array, one uses the total length of the data field instead of using the offset of the next element to calculate its length.

Figure 3.3 is an example of a data field that contains two tags: department id - ”DEID” and sequence number - ”SEQN”. First two bytes of the data field contain `0x0002`, which means that there are two tags contained in the data field. Next eight bytes (bytes 2-9 on Figure 3.3) contain the ”DEID” and ”SEQN” UTF-8 encodings. Bytes 10 - 13 is an offset array that contains two offsets: 14 for the first tag in tags array (”DEID”) and 16 for second tag (”SEQN”).

Class `DataFormatter.java` provides the described data formatting functionality. It consists of 3 static methods:

```
protected static byte[] costructPayload(ArrayList<Pair> elements)
protected static ArrayList<Pair> interpretPayload(byte[] payload)
protected static byte[] getRawData(ArrayList<Pair> elements)
```

`costructPayload` method is a method for constructing data fields according to the described format. It takes in `ArrayList<Pair>`, where each pair is `String` tag - `byte[]` data pair and returns the resulting `byte` array. To get the data field in Figure 3.3, we would need to run the following code (assuming ”DEID” is `0x0001` and ”SEQN” is `0x0002`):

```
ArrayList<Pair> payloadData = new ArrayList<>();
payloadData.add(new Pair("DPID", new byte[] {(byte)0x00, (byte)0x01}));
payloadData.add(new Pair("SEQN", new byte[] {(byte)0x00, (byte)0x02}));
byte[] result = DataFormatter.costructPayload(payloadData);
```

Method `interpretPayload` is just doing the reverse operation of interpreting `byte` into `ArrayList<Pair>` with tags and their data. `getRawData` method gets the raw bytes of data from a data field (e.g. on Figure 3.3 that would be bytes 14-17) and is used for constructing the certificate.

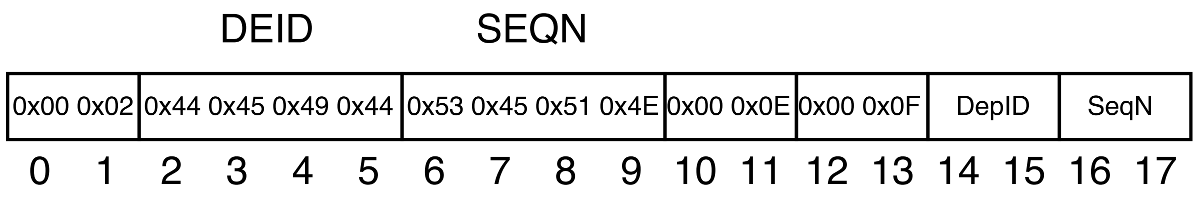

Figure 3.3: Message example

The advantages of using a simple structure like this instead of raw data are quite obvious. First of all the contents of a data field are easier to read, even if we dont know in advance what is inside. Secondly, the data does not have to be communicated in consistent order the tags make sure that even if the order is changed, it never goes unnoticed by the receiving party, and hence the data in the data field will not be misinterpreted. And finally using this protocol assures, that even if sender of the APDU includes some new and unexpected elements in the data field, the receiver will still be able to interpret the it if all of the expected elements are still in place.

3.5 Assymetric card authentication protocol

The project can be divided into three major components: application that runs on the card issuing device, application that runs on the door controller and applet that is installed on the card. Before starting to operate a card needs to be initialized by the issuer. The issuer terminal assumes that the protocol applet is already installed on the card.

3.5.1 Card initialization and operation of the issuer terminal

The card initialization process is a message exchange between card and the issuer terminal, necessary for further operation of the card. The scheme of the messages can be seen on Figure 3.4. First of all the issuer terminal creates a pair of Elliptic Curve Digital Signing Algorithm (ECDSA) keys that are used for signature and verification of the certificate. To start communication with the applet the issuer needs to send the select command APDU with the AID of the applet to the JCRE. After a select command is successfully executed and until power down or deselect command all APDUs that are received by JCRE will be forwarded to the protocol applet. At the next step, the issuer terminal sends the command to the card to generate a pair of ECDSA keys. As the response, the card sends the encoded public key, wrapped in the standard explained earlier. After the terminal receives the pubic key from the card it creates the certificate. The certificate consists of two parts: payload and signature. The payload consists of the CRSID of the card holder, card's encoded ECDSA public key, the group that the student belongs to and, finally, date and time of the certificate expiry. The group is used to specify the access rights of particular student. For example a group called Computer science undergraduates of St

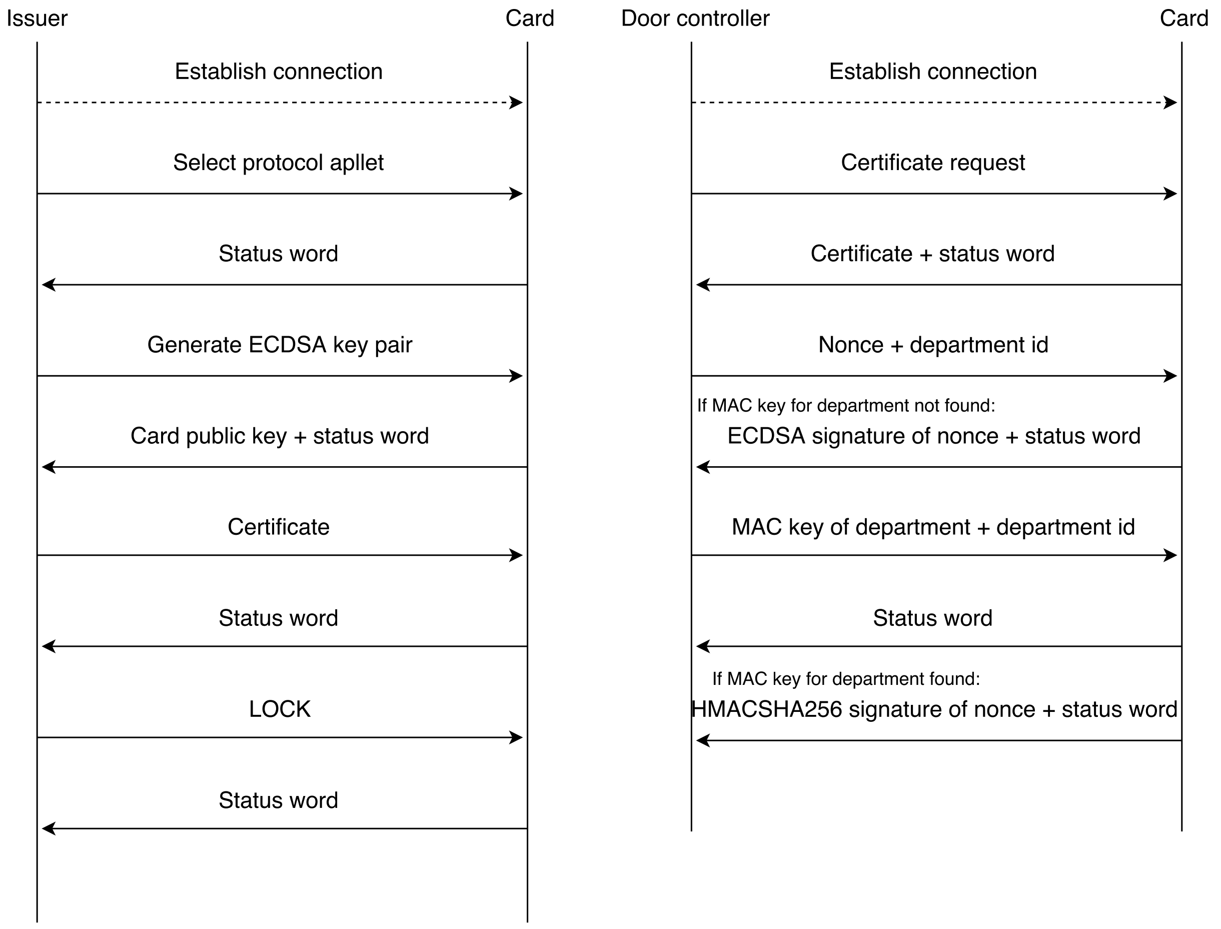
Issuer
Card
Door controller
Card
Establish connection
Select protocol apllet
Status word
Generate ECDSA key pair
Card public key + status word
Certificate
Status word
LOCK
Status word
Establish connection
Certificate request
Certificate + status word
Nonce + department id
If MAC key for department not found:
ECDSA signature of nonce + status word
MAC key of department + department id
Status word
If MAC key for department found:
HMACSHA256 signature of nonce + status word

Figure 3.4: Scheme of the protocol

Johns will all share the same access rights: the Computer Laboratory, college facilities and university library. The signature part of the certificate is the signature of the raw bytes of the payload, signed with the issuer terminal signing key (private `ECDSA` key stored on the issuer terminal). Hence to verify the certificate, the device only needs to know the issuer terminal's public `ECDSA` key. After the certificate is created, it is sent to the card, which stores it until it receives the new certificate from the issuer terminal. After the upload, the terminal verifies that the upload was completed correctly by requesting the certificate from the card and double-checking each field. When all these steps are done, the command is sent to lock the card. After the lock command received, the card stops reacting to some commands including command to store certificate (so the malicious terminal is not able to overwrite the certificate with fake one), command to generate the `ECDSA` key pair (again so that the key that the card uses corresponds to the one that is stored in certificate). Folloqing this, the card initialization process is done.

3.5.2 Operation of the door controller terminal

Door controller application constantly scans for a card. When the door controller is created it is given a signing key of the terminal (terminal `ECDSA` public key), so it can verify the certificate on each card. As soon as the card comes close enough to be detected by the controller, controller requests the certificate. After receiving the certificate, the door controller verifies that the signature corresponds to the data, checks that the expiry date of the certificate is later than the current date and checks if the group that the student belongs to is authorised to open this particular door. If any of these operations fail the door controller denies access to the card. If both operations succeed the door controller generates 20 byte long nonce and sends it to the card along with the department id. Department id would be the same for all the controllers of particular group (e.g computer lab, or St Johns College). At this stage the card has choice of two possible actions to take. If the card has never been authorised to this department its `MAC` key storage will not contain the key for this department. In this case the card signs nonce with its private `ECDSA` key. If the card has been authorised to this department before it can look up the `MAC` key of the department and sign the nonce with `HMAC-SHA-256` using the `MAC` key obtained from the storage.

The reason for having these two options is because `ECDSA` signature is much more complex task, comparing to `HMAC-SHA-256` signature. Hence it should take notably longer for a card to do `ECDSA`. Providing an option for `HMAC-SHA-256` means that card uses more time consuming `ECDSA` only for the first time the department sees the card, and every once in a while after that, when the `MAC` key of the department is updated. After signing the nonce, the card sends the signature to the door terminal, and terminal tries to verify it.

First, terminal uses its `HMAC-SHA-256` key and if not successful, it then attempts to verify the signature with `ECDSA` using card's public key. If any of two verification attempts succeed, the door grants access to the card. If the message was verified using `ECDSA`, the door controller sends card its `MAC` key along with department ID and sequence number of the key. Card stores all this data in its key storage, so the next time it is able to use shorter `HMAC-SHA-256` algorithm for signing rather than `ECDSA`. The department might decide to refresh its `MAC` key every once in a while. In case if it does so, the sequence number will help the card to update its current key, if the key with the later sequence number is received.

3.5.3 Operation of the smart-card

The smart-card applet consists of few classes including KeyStorage class for storing `MAC` keys of the departments, PayloadInterpreter class that retrieves a needed tag from the APDU message that was constructed using a protocol I used for communication and class with my implementation of `HMAC-SHA-256` according to FIPS 198[11]. Key Storage is nothing else than an array of fixed size, that is stored in persistent memory of the card. Every time the key for a particular department is requested, the card scans through the departments that are currently in the array and returns the key if a department was found. Each key in the array has the usage counter, that is incremented every time particular key is requested, so when the card is required to store the new key, it deletes the least used key from the storage, and stores the key for new department instead. The main class of my applet has a very simple structure of big switch statement, that calls different methods based on command APDU received.

3.5.4 HMAC-SHA-256 implementation

As it was explained before, different models of Java Cards have different sets of cryptographic algorithms implemented as a part of the standard library. NXP J3A040 does not have any implementation of the `HMAC-SHA-256` algorithm, so I had to implement it myself according to the FIPS 198[11]. `HMAC-SHA-256` uses blocks of 512 bits each, which is the size that SHA256 hash operates on. The length of the output is again defined by the underlying hash algorithm, and hence it is 256 bits or 32 bytes. The key length that is used is 256 bits.The full description of the algorithm can be found on FIPS 198.

As mentioned before I did not account for the keys of different size in my implementation, I assumed that they are always 32 bytes long. FIPS 198, where the `HMAC-SHA-256` implementation specified does not recommend using keys that are shorter than the underlying hash function output, as they reduce security strength. Having keys longer than the hash output length do not significantly increase security strength.

Potential problems with my implementation of HMAC-SHA-256 As for security reasons I assure that the `MAC` key generated for each of the department is exactly 256 bit long, I did not implement the full handling of shorter or longer keys, hence my implementation does not exactly correspond to RFC specification. Also my implementation at some point uses transient memory of the card to store the key. This may be a security issue, however realistically there is no way that this can be avoided without hugely complicating the algorithm, or overusing the cards persistent memory.

Chapter 4

Evaluation

4.1 APDU exchange logs

4.1.1 APDU exchange between a card and the issuer terminal

The log of APDUs shown in Listing 4.1 demonstrates the initialisation process of the card according to Figure 3.4. All the APDUs are structured in compliance with ISO/IEC 7816-4 (Figure 2.2 and Figure 2.3). The first command (line 2) is a standard `Select AID` command. In the example above my applet has AID of `0xD4D4D4D4D4D40404`. The response APDU `0x9000` (line 4) is "No Error" status word, which means that the command has been successfully executed without errors. Now JCRE will redirect any of the APDUs directly to the applet `0xD4D4D4D4D4D40404`, until the card resets or the deselect command is sent. The next command is `Generate Keys`, which returns a `ECDSA` public key encoding if the keys are successfully generated followed by a "No Error" status word (line 12-14). The terminal then produces the certificate and sends it to the card (line 20-26). The Data Field of the generated certificate is structured according to the Data Field formatting discussed earlier in Section 3.4. The card again responds with a status word, indicating that the certificate was successfully stored. In the next command the issuer terminal retrieves the certificate and verifies that it was stored without error (line 32-38). If we compare lines 20-26 to 30-32 we can see that they are identical apart from the fact that the retrieved certificate is appended with the status word at the end, as it is required by ISO. For the final Command APDU the issuer terminal sends the `Lock the card` command. The card responds with the status word.

```
Command APDU      SelectApp:
00 A4 04 00 08 D4 D4 D4 D4 D4 D4 04 04
Response:
90 00
Command APDU      Generate keys:
80 30 00 00
Response:
90 00
Command APDU      Get Public Key:
80 31 00 00
Response:
04 6E 13 A8 6B 3E 76 3D C7 3B 59 69 CA 64 08 23 10 CA C1 45 5D 4D EA E2
56 CE 17 8B 9B 4F F9 6B 21 AA 4D FD 2B 0D 86 B1 AF 66 14 22 55 15 07 F2
9F 90 00
Command APDU      Reset key storage:
80 35 00 00
Response:
90 00
Command APDU      Store Certificate:
80 33 00 00 94 00 04 43 52 49 44 44 41 54 45 50 4B 45 59 53 49 47 4E 00
1A 00 1F 00 2B 00 5C 64 6E 33 32 33 31 38 30 35 32 30 31 38 31 33 30 38
04 6E 13 A8 6B 3E 76 3D C7 3B 59 69 CA 64 08 23 10 CA C1 45 5D 4D EA E2
56 CE 17 8B 9B 4F F9 6B 21 AA 4D FD 2B 0D 86 B1 AF 66 14 22 55 15 07 F2
9F 30 36 02 19 00 E5 AE 63 E0 61 A7 F3 5E 33 44 60 5F ED BB 2C A0 13 3D
D3 1A 28 C6 E2 41 02 19 00 8D 18 EB AC B6 8C D3 10 3C C3 83 1B 5A 22 03
E1 9F E6 21 DA 4D 29 FE C3
Response:
90 00
Command APDU      Retrieve certificate:
80 34 00 00
Response:
00 04 43 52 49 44 44 41 54 45 50 4B 45 59 53 49 47 4E 00 1A 00 1F 00 2B
00 5C 64 6E 33 32 33 31 38 30 35 32 30 31 38 31 33 30 38 04 6E 13 A8 6B
3E 76 3D C7 3B 59 69 CA 64 08 23 10 CA C1 45 5D 4D EA E2 56 CE 17 8B 9B
4F F9 6B 21 AA 4D FD 2B 0D 86 B1 AF 66 14 22 55 15 07 F2 9F 30 36 02 19
00 E5 AE 63 E0 61 A7 F3 5E 33 44 60 5F ED BB 2C A0 13 3D D3 1A 28 C6 E2
41 02 19 00 8D 18 EB AC B6 8C D3 10 3C C3 83 1B 5A 22 03 E1 9F E6 21 DA
4D 29 FE C3 90 00
Command APDU       Lock the card:
80 38 00 00
Response:
90 00
```

Listing 4.1: APDU exchange between a card and an issuer terminal

4.1.2 APDU exchange between the card and the door terminal

```
 1  Command APDU      SelectApp:
 2  00 A4 04 00 08 D4 D4 D4 D4 D4 D4 04 04
 3  Response:
 4  90 00
 5  Command APDU      Retrieve certificate:
 6  80 34 00 00
 7  Response:
 8  00 04 43 52 49 44 44 41 54 45 50 4B 45 59 53 49 47 4E 00 1A 00 1F 00 2B
 9  00 5C 64 6E 33 32 33 31 38 30 35 32 30 31 38 31 33 30 38 04 6E 13 A8 6B
10  3E 76 3D C7 3B 59 69 CA 64 08 23 10 CA C1 45 5D 4D EA E2 56 CE 17 8B 9B
11  4F F9 6B 21 AA 4D FD 2B 0D 86 B1 AF 66 14 22 55 15 07 F2 9F 30 36 02 19
12  00 E5 AE 63 E0 61 A7 F3 5E 33 44 60 5F ED BB 2C A0 13 3D D3 1A 28 C6 E2
13  41 02 19 00 8D 18 EB AC B6 8C D3 10 3C C3 83 1B 5A 22 03 E1 9F E6 21 DA
14  4D 29 FE C3 90 00
15  CERTIFICATE VERIFIED
16  Command APDU      Get Signature:
17  80 32 00 00 16 01 02 05 21 FF 6E 6A 2E 85 0F F3 12 78 69 62 E4 F3 1A 9E
18  79 9F C3
19  Response:
20  30 35 02 19 00 C9 50 A6 29 AC B5 63 21 8D 3B 60 09 57 C1 51 10 6A 43 93
21  A0 35 4A C4 29 02 18 37 5F 54 7F E5 2A D7 A6 A7 A9 B0 4D B0 99 E9 28 F1
22  8C E0 B6 70 73 F7 F6 90 00
23  MAC key failed. Using ECDSA
24  ACCCES GRANTED
25  Command APDU      Send mac key:
26  80 37 00 00 38 00 03 44 45 49 44 53 45 51 4E 4D 41 43 4B 00 14 00 16 00
27  18 01 02 00 01 93 74 5C EB 10 93 36 E1 A7 D6 B5 9A 6E 46 C5 B9 3A EC 20
28  9C AB 96 2A ED 60 C3 6C 84 3C CF 82 FA
29  Response:
30  90 00
```

Listing 4.2: APDU exchange between a card and a door terminal

The log of APDUs in Listing 4.2 demonstrates a process of authentication of a card to the door controller emulated on my PC. The first command on line 2 is the traditional `Select AID` APDU. The card responds with ”No Error” status word on line 4. The door requests a certificate on line 6 gets it as a response from the card on lines 8-14. The door then verifies the certificate with the issuer terminal's public `ECDSA` and proceeds further, as the verification was successful (line 15). It then sends the card its department id and generated nonce for the signature on line 17-18. The card responds with the signature of the nonce followed by the status word ”No Error”. The door controller tries to verify the signature with `HMAC-SHA-256`, using its `MAC` key in case if the door has been authenticated before. In case the verification was not successful, the door controller tries to verify the signature again, using `ECDSA` and the card's public key retrieved from the certificate. In this case the `ECDSA` verification was successful and door grants access to the card (line 15). Because the failure of the `HMAC` verification indicates that the card does not have the `MAC` key to the department, as the last command the controller sends its key to the card (line26-28). The card responds with the ”No Error” status word, which means that the key was successfully stored on the card.

4.2 Runtime of the authentication

To time different stages of protocol I wrote simple class `Timer.java` that returns timing results in milliseconds.

4.2.1 Runtime using ECDSA

To understand how much on average it takes for the card to authenticate to a door controller, I ran the authentication process 100 times. It resulted in 15066ms of total time, which means that it takes 150.66ms on average for each authentication, with standard deviation of 8.47ms. If we assume that the authentication times are normally distributed, we can estimated the maximum authentication time with a certain confidence level. Figure 4.1 shows 95% left-sided confidence interval for authentication times using an ECDSA signature. As we can tell from Figure 4.1 we can state with 95% confidence, that the authentication should not take longer than 164.5ms.

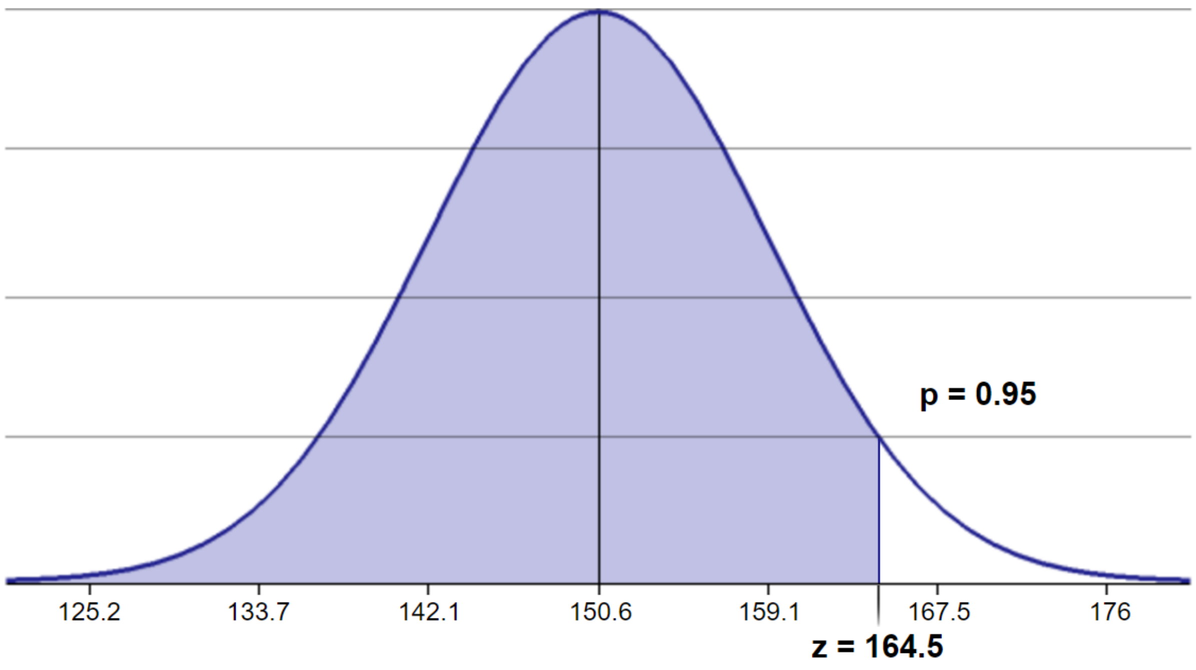

Figure 4.1: Normal Distribution of authentication times using `ECDSA`

4.2.2 Runtime using HMAC-SHA-256

To measure the average length of the authentication process of the card involving the `HMAC` signing and verification I again ran it 100 times. This time my card contained the `MAC` key to the department of the door controller. The results were surprising; it took 15788ms in total, which means that each authentication is on average 157.88ms long with a standard deviation of 7.89ms. This is 7.22ms longer on average than in the case of using an asymmetric signature with `ECDSA`, which was not initially expected. Building the bell curve for this set of data (Figure 4.2), let's us state with confidence of 95% that authentication using HMAC-SHA-256 will not take longer than 170.8ms.

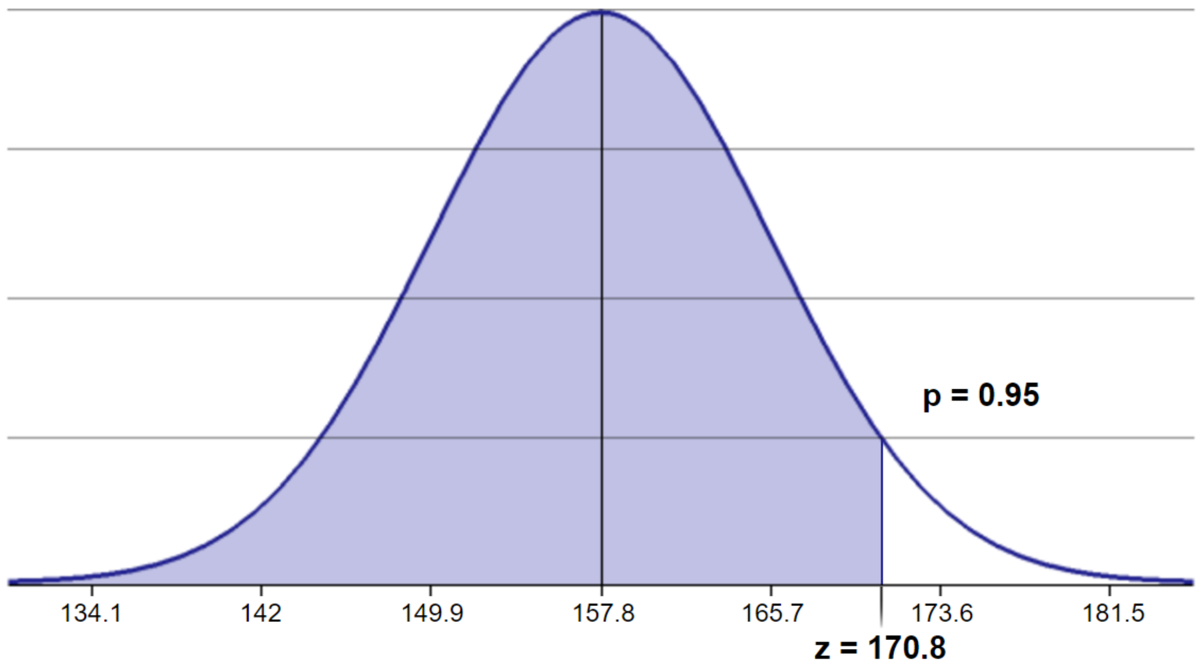

Figure 4.2: Normal Distribution of authentication times using HMAC-SHA-256

As I stated earlier, we usually would expect asymmetric cryptography to be much slower than symmetric. In my project, however, the implementation of HMAC-SHA-256 was written by me, and not taken from the standard Java Card library (J3A040 does not support any kind of HMAC). This shows that algorithms in Java Card standard library are highly optimized.

Overall we can make the conclusion that for the cards that do not have an implementation of HMAC algorithm and are using the protocol described in this dissertation it is faster to only use ECDSA asymmetric cryptography for authentication.

4.2.3 Comparison to MIFARE

If we compare the timings acquired from measuring the performance of the asymmetric authentication protocol implemented in this dissertation with timings from MIFARE Classic data sheet[12], we can see that asymmetric authentication is much slower. The MIFARE Classic authentication procedure, according to the data sheet, takes only 2ms and authentication using the described protocol takes on average 157.88ms in case if ECDSA is used for the signing and verification of the nonce.

4.3 Security analysis of the protocol

4.3.1 Protection against replay attacks

Replay attack is the type of attack on the smart-card when the communication between the reader and the card is recorded by an attacker for the purpose of later usage. The recording can be done in a few ways. The attacker might put some discrete device between the reader and the card during authorisation, similar to ATM skimmers, or step by step recording can be done, when the attacker records first the message from the reader, then later, when it is in operation proximity of a reader it transfers the first message to the card and records its response, so the response can later be transferred to the reader, to get a second message of the authentication and etc.

My protocol is protected against this kind of attack as the first message that comes from a door controller contains nonce that cannot be predicted in advance and is different at every session. If nonce would be shorter than the one I use, it might be feasible for the attacker to attempt to record the card responses to every possible nonce, however in my case it is 20 bytes long, which gives around 10^{48} possible nonces, so it would take around 10^{44} gigabytes just to store every possible nonce, without even considering storing the appropriate responses. If we assume that each challenge-response takes half a second, then to collect responses for all the possible nonces would take approx 10^{14} years. Hence we can conclude that replay attack is impossible if my protocol is used.

4.3.2 Protection against impersonation

One of the biggest flaws of the MIFARE Classic smart-card system is the ease with which the attacker can clone an existing card and impersonate anyone on the system. As it was explained earlier all the attacker needs to have is a cheap Chinese analogue of the MIFARE card with writable UID block, and any MIFARE classic compatible reader (Android phone with NFC technology can be used). This means that the attacker will have the same access as the cloned card has, and on the system hid actions will be completely indistinguishable from the actions of the victim (true owner of the cloned card). The system described in this dissertation have complex protection aimed at preventing cloning attack and subsequent impersonation of some victim by an attacker.

First of all each card generates its own `ECDSA` key pair, which will be unique to each card. Then the issuer signs the certificate that contains owners ID and cards public key with its own `ECDSA` private key, and sends it to the card for storage. Hence in order to clone the card or make the duplicate, the attacker would have to know either cards or issuers private `ECDSA` key. If the attacker somehow gets hold of the issuers secret key than he or she is able to take uninitialized card, that contains the protocol applet, generate the key pair on the card and create a certificate for it, with whatever ID and access rights he or she desires. In case if the attacker manages to get hold of cards private key, than he or she will be able to write an applet that acts similar to the one described earlier, except when the applet is asked to generate the keys, it would just use the victims key pair. In this case the attacker still needs to poses or have access to an issuer terminal with valid issuers `ECDSA` key pair, so the card can receive and store the certificate. Since the secret key never leaves the device that generated it, it should be impossible to get hold of it (of course assuming that the protocol implemented in agreement with common security rules: e.g. not storing key in plaintext).

Another possibility that we should look at is the adversary getting hold of `HMAC-SHA-256` key for one of the departments. In this case an attacker should be able to just copy the certificate from an existing card, and use the `HMAC-SHA-256` key to sign the nonce when the cloned card is authorised to the department, that it holds the key for. Hence the authorisation process does not involve clone card using `ECDSA` to sign the doors nonce, so there is no need for it to know the cloned cards `ECDSA` private key. This will only let the clone card to access the department that it knows `HMAC-SHA-256` key for. As the whole `HMAC-SHA-256` signature is used in my protocol to speed up the process of authorisation, in systems where security is of priority over the speed of authorisation, `HMAC-SHA-256` short-cut can be removed from the protocol, ensuring that only `ECDSA` can be used to authorise to the department.

The protocol described in my work can be further improved by issuer and a card agreeing to the symmetric key, that later will be used by door controllers of the departments to encrypt the `HMAC-SHA-256` key for the department before sending it to the card. So, if the attacker were to eavesdrop on such communication, it would be impossible to restore department key from cypher text. The only way that the attacker would be able to get hold of the department key in this case is to somehow figure out the secret key that was agreed during the initialization of the card.

4.3.3 Cost of the equipment to implement the developed system

The current setup that we currently have in the University of Cambridge. The university uses Mifare Classic 4k cards, each of which comes at the price of £1 [14] when bought in wholesale. However the suspicion is that the university uses some kind of third party consultancy for university card system maintenance, as each card comes to students at a price of £10. The readers used differ in model and manufacturer. The one that is used by st Johns College is HID SmartID. It is available available at the price of around £60 for unit[15]. The system described in this dissertation uses NXP J3A040 microprocessor cards, that are available for a wholesale price of around £3 for unit[16]. Any device that implements PC/SC and supports ISO 1443 can be used as a door controller. Almost all smart-card readers available on the market nowadays are PC/SC compatible and their price ranges from £30 to a few hundreds. PC/SC Single Door module can be an example of device that can be used for the new system. It has a 32 bit CPU and Open Systems Operating System, so it can carry out all the necessary computation on its own. Hence you can see that the new system will require changeover expenses, as all the readers and cards will need to be swapped, but overall the new system will cost around as much MIFARE Classic implementation that is currently in use with an advantage of being much more secure.

4.3.4 Weaknesses of the protocol and attacks on it

Relay Attack

One of the possible attacks that can be performed on the implemented protocol is man-in-the-middle-attack. Even though the overall conditions for success of this attack are very complex, the use of contactless technology makes it somewhat easier to perform. To execute this type of attack, the attacker needs to have two devices that are connected together (for example through the internet), one of which emulates the reader, and the second emulates the card.

The device that emulates the reader is placed in operating proximity of the victims card. For usual RFID door controllers the operating proximity of the device is few centimetres. However it is possible to build the device with stronger field that is able to power up the card from the distance of up to 1 metre.

The device that emulates the card is placed in operating proximity of the door terminal that is targeted by the attack. As soon as the door terminal begins communication with the attackers card emulator device, it transfers all the messages to the door terminal emulator near the victim.

These two devices basically make the victims card believe that it talks to the door controller and target door controller believe that it talks to the victims card. Hence the door controller grants the access to the attacker, even though the actual card that participated in authentication process may be tens of kilometres away from the targeted door controller. The great complication in the way of the attacker is the fact that both devices must be in proximity of both target door controller and victims card at exactly same moment, otherwise the attack will not work. However with a high power RFID reader, that is placed in a place where a lot of people with targeted systems cards go through can simplify this process (entrance to university library). In this case, the attacker just needs to hold the card next to the door controller until someone enters the library with the smart-card. Similar technique can be used to make contactless payments with bank cards without authorisation of the user.

Chapter 5

Conclusion

This project contains development and implementation of the prototype door access control system based on asymmetric cryptography using Java Card platform. As one of the preparation steps I have explored the weaknesses of the existing MIFARE Classic system, that is currently used in the University of Cambridge and many other places, in an attempt to understand what distinguishes weak security protocol from the strong and avoid mistakes made by the developers of MIFARE Classic.

5.1 Achievements

I have fulfilled the success criteria of the project, as the designed protocol is protected against cloning, brute-force, replay and impersonation attacks. The authentication process takes no more than 170 milliseconds, which is well below the goal of making a protocol that authenticates the card in less than a few seconds. As the end result of my project, the command line interface was created to implement the functionality of an issuer terminal that is able to initialize new cards with the CRSid of the person, and a door controller that decides whether to grant or deny access to a particular card.

This dissertation apart from describing the development of asymmetric authentication protocol, can also be used as a step-by-step guide for someone, who is intending to learn basics of Java Card development, as the information that it contains would greatly lessen the learning curve.

5.2 Lessons learned

During the implementation of the protocol I became familiar with Java Card technology and development of Java Card applets. This included acquiring an understanding of the peculiarities associated with the usage of different types of memory (persistent and transient) available on smart-card, developing flexible format for sending data in APDU data field and programming the implementation of `HMAC-SHA-256`. The prototype did not work fully as designed, due to the fact that instead of using `HMAC-SHA-256` that I expected would be included as part of a Java Card standard library, I had to implement it myself, which made it unexpectedly slow. This fact demonstrated how big a difference hardware and software optimisation can make. As a result, the part of the protocol that was designed to speed up the overall authentication process actually slowed it down by a few milliseconds. However, if protocol is run on a smart-card that has an implementation of `HMAC-SHA-256` as a part of a standard library, it should perform as intended.

This project has been proven to have a very steep learning curve due to the lack of information on Java Card applet development process and absence of a readily available framework. If starting again with the benefit of hindsight, I would be able to move much faster through the initial and, undoubtedly, the most difficult stages of the project, where a lot of time was spent on research and trial-and-error process.

5.3 Further work

- Improve the authentication scheme to provide mutual authentication of a card and controller, instead of just the door authenticating the card.
- Improve the authentication scheme to provide user untraceability.
- Optimize the `HMAC` implementation, so the cards that do not have it implemented as a part of a standard Java Card library can benefit from faster symmetric authentication.
- Create a graphical user interface for issuer terminal.

Bibliography

[1] Finkenzeller Klaus, *RFID handbook*, WILEY, Third Edition, 2010.

[2] Nohl Karsten, David Evans, Henryk Pltz, *Reverse-engineering a cryptographic RFID tag*, Radboud University, 2008.

[3] Nicolas Courtois *Card-Only Attacks on MiFare Classic*, http://www.nicolascourtois.com/papers/mifare_all.pdf

[4] *GPShell*, https://sourceforge.net/p/globalplatform/wiki/GlobalPlatform%20Card%20Specification/

[5] *Global Platform*, https://www.globalplatform.org/

[6] *PC/SC*, https://www.pcscworkgroup.com/

[7] *PC/SC*, https://www.pcscworkgroup.com/

[8] *JavaCardPro*, https://javacard.pro/

[9] *JCAlgTest*, https://github.com/crocs-muni/JCAlgTest

[10] Ankur Mittal *Roughtime protocol*, https://roughtime.googlesource.com/roughtime

[11] National Institute of Standards and Technology *FIPS 198*, http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf

[12] National Institute of Standards and Technology *FIPS 198*, http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf

[13] Giantec Semiconductors *1K bytes EEPROM Contactless Smart Cards comparison sheet*, http://www.proxmark.org/files/Documents/13.56%20MHz%20-%20MIFARE%20Classic/MIFARE%20Classic%20cles/Giantec_Semiconductor_Inc_GT23SC4439_1K_bytes_EEPROM_Contactless_Smart_Card.pdf

[14] *amazon.co.uk*, https://www.amazon.co.uk/Blank-MIFARE-Classic%C2%AE-4k-Cards/dp/B01COAAV16/ref=sr_1_11?ie=UTF8&qid=1495181685&sr=8-11&keywords=mifare+classic+4k

[15] *HID iClass*, http://www.ebay.co.uk/itm/HID-smart-id-S10-Mullion-Reader-/252878006200?hash=item3ae0b427b8:g:wkYAAOSwvKtY9p2X

[16] *JCOP J3A040*, https://www.smartcardfocus.com/shop/ilp/id~760/j3a040-cl/p/index.shtml

Appendix A

Project Proposal

Computer Science Tripos – Part II – Project Proposal

Efficient asymmetric cryptography for RFID access control

D. Natykan, St John's College

dn323

21 October 2016

Project Supervisor: Dr M. Kuhn
Director of Studies: Dr R. Mullins
Project Overseers: Dr D. J. Greaves & Dr J. Daugman

A.1 Introduction

The MIFARE Classic smart card security system, which was created by NXP Semiconductors (Philips Electronics), utilizes the standard ISO 1443 Type A protocol for communication on frequency 13.56 MHz. Since the release of the system, more than 3.5 billion cards were produced and more than 200 million are still in use today. The University of Cambridge is one of thousands of institutions that decided to rely on this system. MIFARE Classic cards are used as university cards, providing door access control, printing authorisation and even payment systems for meals in colleges.

The MIFARE Classic system has proven to be insecure in a number of ways. One of the main reasons for this is NXP Semiconductors choice of security by obscurity in its cryptography algorithm. In December 2007, two German researchers (Nohl and Plotz) presented the partial reverse-engineering of Crypto-1[1] (encryption algorithm used in MIFARE Classic). Even despite this being incomplete they already discovered some vulnerabilities. Later, when the algorithm was fully reverse-engineered public tools to hack MIFARE Classic have appeared and today it only takes 1 hour[1] and around 40 GBP[2] worth of equipment to get all the secret keys that the particular MIFARE Classic implementation uses.

Another problem of the MIFARE Classic system, and even some of its' successors (e.g. MIFARE Plus), is that they all use a symmetric encryption scheme. For symmetric access control protocols, it is common that all cards and card readers used for a particular application have the same keys for authentication. This means that if one of the cards is compromised and the cards private key is retrieved, the whole system is compromised. One possible solution is using a diversification technique, which means that each card has its own private key shared with the reader. Even though this approach limits the damage from obtaining the private key from a card (the key obtained can be used only to impersonate the particular card it was retrieved from), it does not address the case of obtaining all the keys from the reader, which again will compromise the whole system. Asymmetric key cryptography provides a solution to this problem by having two types of key: private and public. Private keys are kept secret from everyone apart from the owner. Public ones are made publicly available. If Alice wants to communicate with Bob, she encrypts the message with Bobs public key and sends it to him. Bob then uses his private key to decrypt it. If anyone has eavesdropped on the message, they would not be able to decrypt it without knowing Bobs private key. Hence, even retrieving a private key from a smart card or a reader can only result in ability to impersonate the particular device, from which the key was retrieved, not compromising other cards and readers in the system.

The core of my project is to implement a prototype RFID access-control system that may be used as a more secure substitute for the MIFARE Classic system, which is currently in use in the university. After gaining proficiency in existing asymmetric RFID

[1]performing Darkside attack with MFCUK tool

[2]e.g. Identiv SCL3711 Contactless USB Smart Card Reader

door access-control protocols (e.g. recently emerged ones like OPACITY and PLAID), I will be able to decide whether to implement one of them in my prototype or design and implement a new protocol. The University Card RFID system is currently being reviewed and a successful implementation of this project could usefully inform this process.

A.2 Starting Point

I will draw upon the following in my project:

- My project will use knowledge from the Computer Science Tripos modules that I have studied or going to study this year. Security I and Security II will of course be the most helpful ones, but modules like Computer Networks and Principles of Communication should be very useful as well.
- I will use my experience with either Java using JavaCard development environment or C using .NET card environment to implement the encryption protocol for my prototype of the system.

A.3 Success criteria and evaluation

The end goal of my project is to have a working prototype of RFID smart card system that uses asymmetric key cryptography. The cryptographic protocol should be either designed by me or based on the existing protocol, be protected against impersonation, prevent user traceability and be secure against replay attacks. The prototype should be able to sustain attempts to brute force the keys and take a maximum of a few seconds to authenticate (time interval from powering up the card being held to the reader to completing the authentication) the card. Moreover, successfully attacking one reader should not affect any other readers. In the end I plan to have a Command Line Interface (CLI) application that allows initialising and reprogramming of the cards as well as displaying the information from previously programmed cards. As the evaluation of the project I will measure the speed of the authentication of my prototype and provide the cost of the card chip required. Moreover the time it would take to brute-force one of the keys in my prototype can be calculated. All these data can be compared to the similar numbers obtained from the university card system.

A.4 Success criteria and evaluation

- Preliminary reading and familiarization with existing RFID smart card technology.
- Choosing the particular protocol on which to base my prototype.
- Programming the solution and putting it on the card and the reader.
- Wrapping the solution with Command Line Interface and providing commands to write data on a card, and read it.

- Testing of the system.
- Write-up and evaluation.

A.5 Possible extensions

- Look into MIFARE Classic emulation on my prototype. Being able to emulate MIFARE Classic will make my prototype backward compatible with an existing university system. This would greatly simplify the possible migration from the old system to my prototype.
- Mathematical proof of untraceability of the individual using the system, absolute identity hiding and outsider deniability under the given assumptions can be done as an extension of evaluation.
- Writing a graphical UI for programming cards.
- Introduction of the additional features to the prototype of the system that are relevant to the university (rather than having just the authentication of the user, it will also be beneficial to introduce functionality to group users and restrict their rights according to the entitlements of the group and provide a payment application.

A.6 Plan

1. **Weeks 1-2 21/10/16 4/11/16 Michaelmas Term**
   - Reading and research of the subject. This will include the revision of some aspects of Security I Tripos module, as well as familiarization with deeper concepts specific to RFID smart card security.
2. **Weeks 3-4 5/11/16 18/11/16 Michaelmas Term**
   - Understanding the weaknesses of MIFARE Classic and reproducing some of the known attacks on it will help me to get a better understanding of the requirements of the protocol on which I will base my prototype.
   - Exploration of the available smartcard operating systems (e.g. G&D SmartCafe, NXP JCOP 41, Infineon JTOC), high-level languages and development platforms. Obtaining chosen NFC reader, SDK and empty smart cards of chosen OS.
   - Experimenting with this apparatus.
   - *At the end of week 4, I should have a strong understanding of how the university card system works and begin to gain an understanding of how to operate with the chosen smart card SDK.*
3. **Weeks 5-6 19/11/16 2/12/16 Michaelmas Term**

- Familiarization with the chosen smartcard development platform by implementing a very basic challenge-response authentication protocols, based on both symmetric (MAC) and asymmetric (signature) cryptography.
- Exploration of existing new smart card protocols such as OPACITY and PLAID.
- Choosing the one that is most suited for the success criteria and the goals set to the project.
- *At the end of week 6, I should have decided on whether to use existing smart card protocol for my prototype or to invent one based on the existing ones.*

4. **Weeks 7-10 3/12/16 30/12/16 Christmas Vacation**
    - Final decision on the protocol that the prototype will be based on. It will possibly involve the design of the new protocol.
    - Getting started on coding the encryption scheme for both reader and card.
    - Continue coding the encryption scheme.
    - *By the end of the vacation I should have written the majority of the code for my prototype.*
5. **31/12/16 27/1/17 Christmas Vacation and Lent Term**
    - Contingency time set aside for any issues found up to this point, which can be used for implementing extensions.
6. **Weeks 11-12 28/1/17 10/2/17 Lent Term**
    - Finish coding the prototype.
    - Wrapping it into the Command Line Interface application for easier use and testing.
    - *By the end of week 12, I should have a working prototype of the system wrapped into CLI application.*
7. **Weeks 13-14 11/2/17 24/2/17 Lent Term**
    - Testing of the system and bug-fixes.
8. **Weeks 15-16 25/2/17 10/3/17 Lent Term**
    - The evaluation of the system. The write-up of the dissertation.
9. **11/3/16 24/3/17 Easter Vacation**
    - Contingency time set aside for any issues found up to this point, which can be used for implementing extensions.
10. **Weeks 16-18 25/3/17 7/4/17 Easter vacation and Easter Term**
    - Finishing the dissertation.

- *By the end of week 18, I should have a finished draft of the dissertation and after that only minor tweaks should be done to its contents.*

11. **Weeks 19-20 8/4/17 21/4/17 Easter Term**

    - Final corrections and submission.

A.7 Resources Declaration

For this project I will need a few programmable smartcards with ISO 14443 contactless interface (e.g. Java Card with GlobalPlatform API, such as NXP JCOP,ACOSJ) and a Contactless USB smart card reader compatible with libnfc (e.g. SCL3711). I also need software development kit for the cards operating system. For the programming and documentation, I will be using my own laptop (MSI GS70 2QE) running Windows 10. I will keep backups locally, on an external hard-drive and on cloud storage. For code backup and source control, I will use Git as well as plaintext copies on the external hard drive. In case of the failure of my main computer, I have a backup Surface 3 that will be able to fully replace my main computer. I accept full responsibility for these machines and I have made contingency plans to protect myself against hardware and/or software failure. I intend to release all source code and documentation (dissertation included) as open source after my dissertation is marked.

Bibliography

[1] Henryk Pltz, "Mifare Classic Eine Analyse der Implementierung", *Humboldt-Universitt zu Berlin*, 2007